TSP Simulated Annealing Análise de complexidade

O problema do caixeiro viajante (em inglês abreviado como TSP) consiste em achar a rota mínima entre n cidades de forma que se visite cada cidade apenas uma vez e termine a rota na cidade onde ela iniciou.

Foi utilizado um algoritmo de Simulated Annealing (SA) para resolver esse problema. Para analisarmos a função tempo do algoritmo é preciso primeiro ter a complexidade das funções básicas e depois ver como essas funções se comportam na função principal.

Temos a função startAnnealing como a principal. Ela tem como parâmetros um vetor v, que representa a rota inicial, e uma matriz de custo onde está indicado o custo para ir de uma cidade i para uma j.

Como funções básicas temos:

- **float custo(v, matrizCusto)**, onde v é a rota e matrizCusto a matriz de custo entre as cidades. É responsável por calcular o custo de uma rota;
- As funções **void swap (v, posA, posB)**, **void inverse(v, posA, posB)**, **void insert (v, posA, posB)** e **void changeSolution(v, posA, posB)**, onde v é a rota, posA e posB são índices do vetor v. Essas funções são responsáveis por gerar possiveis novas rotas;
- **float comparacao(v, possivelEstado, matrizCusto),** onde v é a rota, possivelEstado uma rota candidata a ser melhor que a atual e matrizCusto a matriz de custo entre as cidades. Essa função serve para calcular a diferença de custo da rota atual para a possível nova rota (deltaC;
- **float probabilidadeAceitar(temp, deltaC),** ambos parâmetros são números reais usados para calcular a chance do algoritmo aceitar uma solução pior do que a atual.

Análise funções básicas:

float custo(v, matrizCusto)

```
soma = 0                    // 1
para cidadeAtual = 0 até (cidadeAtual < tamanho vetor V)            //n
        soma += matrizCusto[v[cidadeAtual]] [v[cidadeAtual + 1]]    //1
return soma
```

Essa função calcula o custo da rota v e não há melhor ou pior caso.

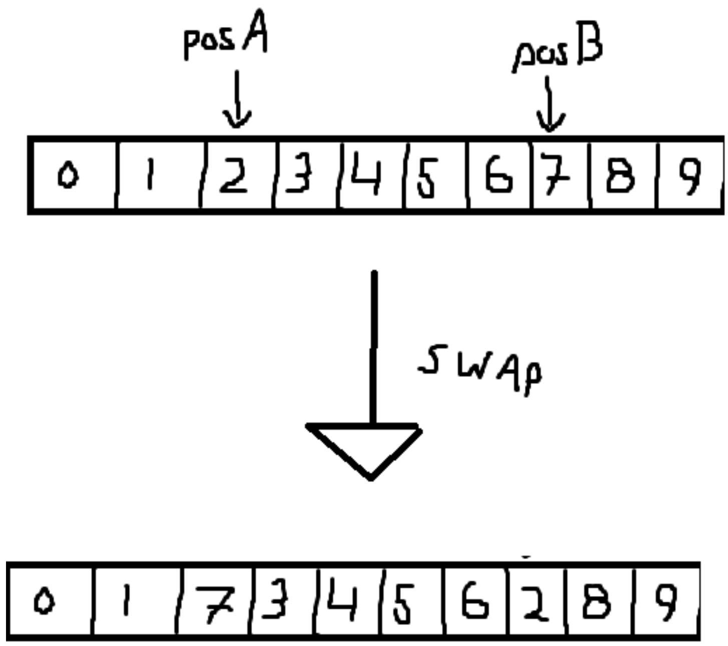
Complexidade: $T(n) = n + 1 + 1 = n + 2 \therefore O(n)$

void swap (v, posA, posB)

```
aux = v[posA]           // 1
v[posA] = v[posB]       // 1
v[posB] = aux           //1
```

Essa função troca dois elementos da rota, e não há melhor ou pior caso, todos os casos tem a mesma complexidade.

Ex:

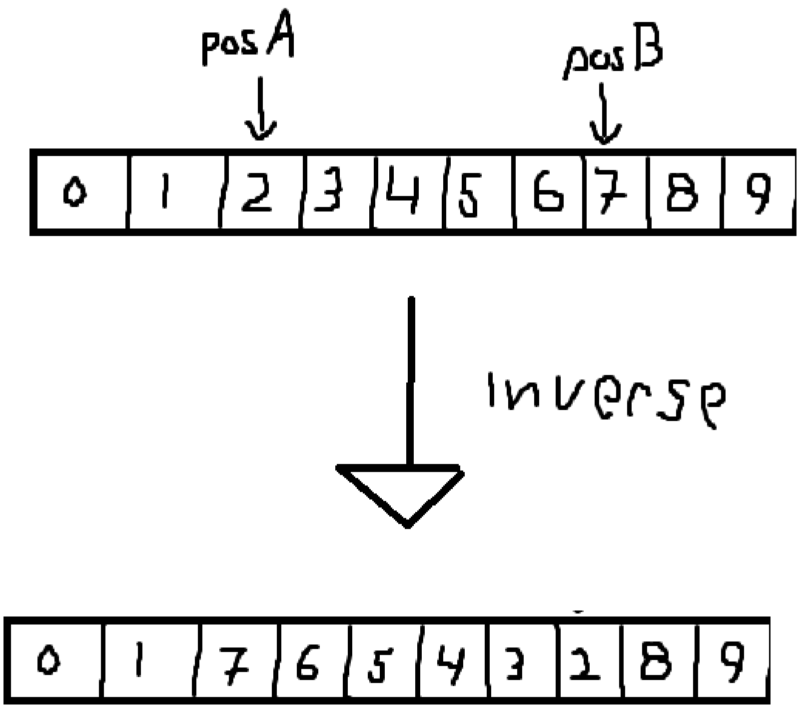


Complexidade: $T(n) = 1 + 1 + 1 = 3 \therefore O(1)$

```
void inverse (v, posA, posB)
    aux                 // 1
    se (posA<posB)          // 1
        para posA até posA<posB     // n/2 ( I )
            aux = v[posA]           //1
            v[posA] = v[posB]       //1
            v[posB] = aux           //1
            posB–                   //1
    senão                           // mesma coisa do caso de cima
        para posB até posB<posA
            aux = v[posA]
            v[posA] = v[posB]
            v[posB] = aux
            posA--
```

Essa função inverte o vetor v da posA até a posB. O pior caso dessa função ocorre quando precisamos inverter de v[1] até v[tamanho], pois é p caso onde teremos o maior loop.

Ex:



Complexidade:

(I) O loop é n/2 porque a medida que o posA incrementa o posB decrementa (e vice-versa) então o loop termina assim que os dois “se encontram” no meio do vetor.

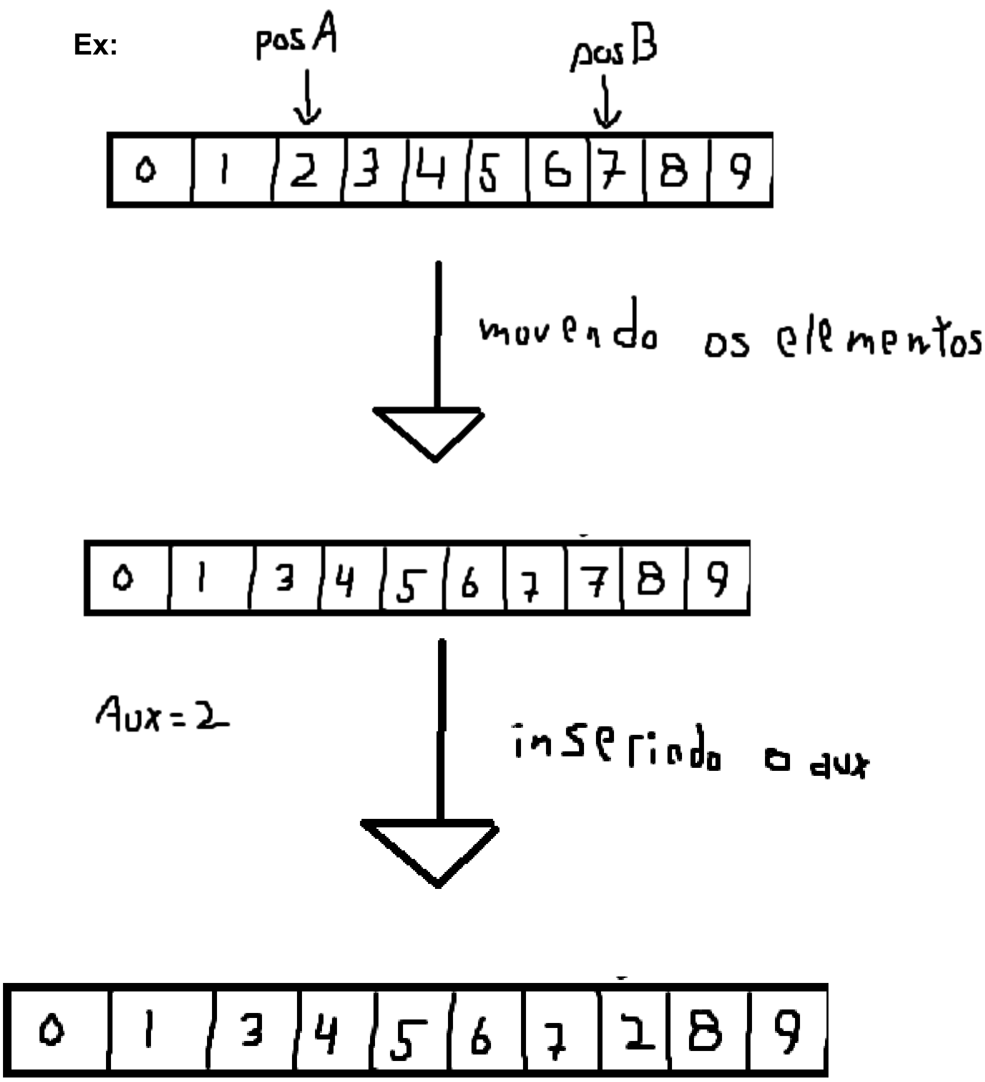
$$T(n) = \sum_{1}^{n/2} (1 + 1 + 1 + 1) + 1 + 1 = \sum_{1}^{n/2} (4) + 2 = (4 * (n/2)) + 2 = 2n + 2 \therefore O(n)$$

void insert (v, posA, posB)

```
    aux                 //1
    aux = v[posA]              //1
    se (posA < posB)        //1
        para i = posA até (i<posB)                  //n
            v[i] = v[i + 1]               //1
         v[posB] = aux             //1
    senão              //1
        para i = posA até (i > posB) passo i–             //n
            v[i] = v[i - 1]                //1
         v[posB] = aux                      //1
```

Essa função insere o elemento do índice posA no lugar do índice posB reorganizando o vetor movendo os elementos para esquerda ou direita. O pior caso ocorre quando inserimos o elemento do índice 0 no lugar do índice n.

Ex:



Complexidade: $T(n) = \sum_{1}^{n}(1) + 1 + 1 + 1 + 1 = n + 4 \therefore O(n)$

void changeSolution(v, posA, posB)

```
    func = random(0,2)     //sorteia um número entre 0 e 2        // 1

     switch(func)
          case 0:
             inverse(v, posA, posB)          //n
          case 1:
             swap(v, posA, posB)             //1
          case 2:
             insert(v, posA, posB            //n
```

Como estamos fazendo a análise de pior caso e todas as funções tem a mesma chance de serem escolhidas, tomamos a complexidade dessa função como sendo a mesma da 'pior' função:

Complexidade: $T(n) = 1 + 2n + 2 = 2n + 3 \therefore O(n)$

float comparacao(v, possivelEstado, matrizCusto)

return custo(possivelEstado, matrizCusto) - custo(v, matrizCusto) //2 * T(n) da função custo

Complexidade: $T(n) = 2 * (n + 2) = 2n + 4 \therefore O(n)$

float probabilidadeAceitar(temp, deltaC)

return $e^{-deltaC/temp}$ //1

Complexidade: $T(n) = 1 \therefore O(1)$

Agora após analisarmos todas as funções básicas podemos analisar o T(n) da função startAnnealing.

int* startAnnealing(v, matrizCusto)

```
    rotaMin = new int tamanho            //1
    rotaMin = v                          //n
    custoMin = custo(v, matrizCusto)     //n +2

    possivelEstado = new int tamanho     //1
    possivelEstado = v                   //n
```

enquanto (temperatura>1) $//log_{\alpha^{-1}}^{temperatura}$ (I I)

```
        para i = 0 até (i<loopInterno)          //n
              sorteia posA e posB               //1
```

```
                    changeSolution(possivelEstado, posA, posB)                    //n
                    deltaC = comparacao(v, possivelEstado, matrizCusto)        //2n +4
                    se deltaC<0                        //1
                              v = posssivelEstado   //n
                    senao              //1
                              se random (0, 1) < probabilidadeAceitar(temperatura, deltaC)
//1
                                        v = possivelEstado //n


                    se custoMin > custo(v, matrizCusto)  //n + 2
                              rotaMin = v                   //n
                              custoMin = custo(v, matrizCusto)                  //n + 2

          temperatura  = temperatura * alpha              //1
```

(I I)

Repetições	Temperatura
1	temp
2	temp*alpha^2
3	temp*alpha^3
4	temp*alpha^4
5	temp*alpha^5
...	
i	1

Então, seguindo o padrão temos que o critério de parada desse loop ocorre quando:

$$\frac{temp}{\alpha^{-i}} = 1 \rightarrow \quad temp = \alpha^{-i} \rightarrow temp = (\frac{1}{\alpha})^{i} \rightarrow i = log_{\alpha^{-1}}^{temp}$$

Complexidade:

$$T(n) = 1 + n + n + 2 + 1 + n + \sum_{1}^{log_{\alpha^{-1}}^{temp}} [1 + \sum_{1}^{n}(n + 1 + n + 2n + 4 + 1 + n + 1 + 1 + n +$$

$$n + 2 + n + n + 2)] = 3n + 4 + \sum_{1}^{log_{\alpha^{-1}}^{temp}} [1 + \sum_{1}^{n}(9n + 12)] + 3n + 4 + \sum_{1}^{log_{\alpha^{-1}}^{temp}} [1 + 9n^2 +$$

$$= 3n + 4 + 9log_{\alpha^{-1}}^{temp}n^2 + 12log_{\alpha^{-1}}^{temp}n + log_{\alpha^{-1}}^{temp} \therefore O(log^{temp}n^2)$$